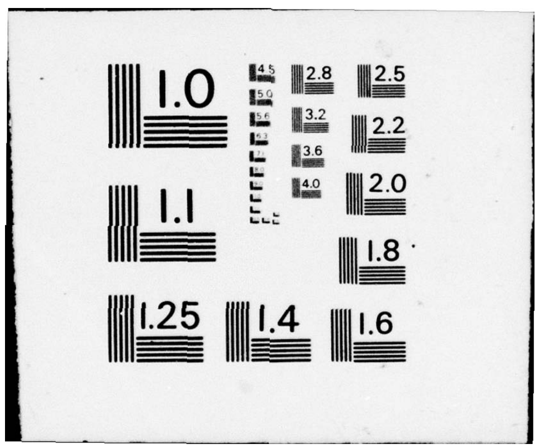
AD-A033 378 MICHIGAN UNIV ANN ARBOR DEPT OF ELECTRICAL AND COMPU--ETC F/G 9/2
DATA FLOW CONSIDERATIONS IN IMPLEMENTING A FULL MATRIX SOLVER W--ETC(U)
SEP 76 D A ORBITS, D A CALAHAN AF-AFOSR-2812-75
UNCLASSIFIED 98 AFOSR-TR-76-1212 NL

1 OF 1
ADA033378

END
DATE
FILMED
2 - 77


1.0
4.5
5.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

AFOSR - TR - 76 - 1212

DATA FLOW CONSIDERATIONS IN IMPLEMENTING A FULL MATRIX SOLVER WITH BACKING STORE ON THE CRAY-1

See 1473

D. A. Orbits

D. A. Calahan

September 1, 1976

Report No. 98

Systems Engineering Laboratory
University of Michigan
Ann Arbor, Michigan 48109

Sponsored by the Mathematical and Information Sciences Directorate
Air Force Office of Scientific Research under Grant AFOSR 75-2812

Abstract

Techniques for the solution of full systems of simultaneous equations represent an important algorithm class for vector processors. This report considers the data flows involved in solving a full equation system on the Cray-1. This involves study of the I/0 and memory-processor path traffic vis-a-vis the capabilities of the Cray-1 to support it. The I/0 is found to present problems for small systems of equations and in the substitution phases of small and large systems. Using an algorithm proposed in the report, the memory-processor path is shown to have excess bandwidth. Suggestions are made for utilizing this bandwidth to increase the arithmetic operation rate by modifying and expanding the processor architecture.

Contents

I. Introduction

The communication links between the major components of the Cray 1 system - the processor, main memory, and mass memory devices (backing store) - are the architectural features most critical to algorithmic development after the vectorization feature itself. Only a single data path exists between memory and processor, so that even though the 16-way interleaved memory has capacity to transfer words at four times the clock speed, only ¼ of this speed is available to the processor. This leaves potenetially 3/4 of the memory bandwidth (i.e., 240 megawords/sec.) for I/0. However, the I/0 channels have capacity for only 12.9 megawords/sec. when operating with a full complement of 24 discs. The potential for a communications bottleneck on both sides of memory is therefore significant.

One can claim that in fact a three-level memory heirarchy exists, by viewing the vector register set as functionally a single or dual cache. However, the register and functional unit speeds are matched, so that no communication problem exists at this end of the memory heirarchy.

Algorithmic remedies for these communication problems are of the general philosophy that computational complexity should be maximized on data at each memory level. If the total arithmetic complexity is independent of data organization in the memory levels, this strategy insures that the fewest data movements are required when maximum computational use is made of data at each level. Practically, this procedure involves vector and small matrix operations like inner- and outer- products on data in the

register cache before sending the result to main memory, and large matrix operations on data in main memory before communication with backing store.

This study will be concerned with examining the data flows associated with solution of a full set of linear simultaneous equations. It happens that, in movement and operations with large data blocks, programming effort is increased to handle incomplete blocks and intrablock computation (e.g., the movement across the pivot elements in sweeping through a column strip of a matrix). However, the major communication problems can be studied by examining two critical loops in an equation solution algorithm: (1) the outermost loop, where megaword blocks are moved between the backing store and main memory to satisfy the gross needs of the functional units on the other side of memory for operands and to store results, and (2) the innermost multiplication-subtraction loop in the solution, where the memory-to-register flow is critical.

Before proceeding, it should be noted that in investigating methods to reduce the impact of an apparent data flow bottleneck, we will not only remove the flow constriction but have in one case by algorithmic means exposed excess flow capacity. This in turn suggests that additional functional unit capacity could be accomodated - either through parallelism or faster pipelines. If similar results could be obtained for a sufficient number of major applications, this would indicate that future architectural redesign and expansion be directed toward the functional units rather than the data paths, heretofore felt to be the most critical architectural feature.

II. Evaluation of the I/O Bandwidth

A. Introduction

It is easy to show that a Cray-1 is capable of solving a large system of n linear simultaneous equations in approximately $4.8n^3$ nanoseconds, including a 10 clock overhead in the innermost loop to divide long vectors into ones of length 64. A system with an 8 megabyte main memory can consequently solve 1000 equations - filling main memory - in 4.8 seconds. Thus, an equation set worthy of vector processing inevitably involves use of a backing store.

Smaller full matrices are often encountered as part of larger sparse matrix solution; these full components reside on a backing store, are loaded at appropriate times during the overall solution process, and undergo a sequence of computations similar to the factorization of a single full matrix. Such matrices are typically in the order of $20 \leq n \leq 100$.

The backing store itself can consist of between 1 and 12 disc units on both input and output, each capable of a transfer rate of 34.5 megabits. This yields a total input capacity of .54-6.48 megawords/second. In the following sections,we will establish that this rate can be inadequate to support the gross computational needs of the arithmetic fuctional units both for small and for very large sets of simultaneous equations, but is adequate for many typical matrix problems of intermediate size.

B. Small matrices initially on backing store

To read a matrix of dimension n with m input channels from backing store requires $1.85\ n^2/m\ \mu s$. This becomes equal to the

solution time ($4.8n^3$ ns) when $n=385/m$ or $n=32$ for full input channel capacity. For smaller values of n, the processor will be busied $nm/385$ of the time, a some what alarming result for the range $20 \leq n \leq 100$, even with full channel activation.

C. Large full matrices

In the LU triangular factorization of a matrix too large to be contained in real memory, it becomes necessary to recall from backing store factored parts of the matrix to participate in operations on a part of the matrix currently being factored. For the sake of discussion, it will be assumed that the matrix is partitioned into column strips, so that factorization takes place on a strip currently in main memory by recalling previously-factored strips.

As an example, in Figure 1, a full matrix of size n (=6) is divided into k partitions of storage S_k, each having $p = n/k$ columns. The number of writes to backing store is n^2, assuming that the entire factored matrix must reside on backing store. The number of reads is

$$
\begin{array}{cccccc}
\multicolumn{2}{c}{\text{kth strip}} & \multicolumn{2}{c}{\text{2nd strip}} & \multicolumn{2}{c}{\text{1st strip}} \\
X & X & X & X & X & X \\
X & X & X & X & X & X \\
X & X & X & X & X & X \\
X & X & X & X & X & X \\
X & X & X & X & X & X \\
\multicolumn{2}{c}{\underbrace{X \quad X}} & \multicolumn{2}{c}{\underbrace{X \quad X}} & X & X
\end{array}
$$

$9\text{x}2 + 5\text{x}1 = 23$ reads

Figure 1. Counting reads in a factorization

$$N_r = \sum_{r=1}^{k-1} \frac{p(p-1)}{2} r + p^2(k-r)^2$$

$$= \frac{p(p-1)(k-1)(k)}{4} + \frac{p^2(k-1)(k)(k-1/2)}{3} \qquad (1)$$

With total storage S and strip size S_k, then $k = S/S_k$ and $p = S_k/\sqrt{S}$. The second term of (1) easily dominates the expression, and this dominant term can be written*

$$N_r \cong \frac{S^2}{3S_k} = \frac{p^2k^3}{3} \qquad (2)$$

With half of main memory devoted to the current and the recalled strips, the fraction σ = (computation time)/(I/O time) can be computed from (2) as

$$\sigma = \frac{3.9 \times 10^3 m}{n}$$

Thus, a single disc could supply operands up to n = 3900, which requires 4.7 minutes of computation time. This appears more than adequate I/O bandwidth.

*Equation (2) also applies to a large class of sparse equations [2].

D. The I/O problem in the substitution process

The I/O problem is proportionally more severe for the forward and back substitution steps, where only a few numeric computations are performed with each element of the recalled factored matrix.

Consider an arithmetic computation sequence where K arithmetic computations are performed on the average on every L words in main memory, by a processor with an operation rate of M floating point operations/second (FLOPS). Then the memory must be supplied from the backing store at the rate of

$$ML/K \text{ words/sec.} \qquad (3)$$

In the forward and back substitution stages, the inner loop instruction will be of the general form

$$X(I) = X(I) - LU(J)*YD \qquad (4)$$

where LU(J) contains the elements of $\underline{L}$ and $\underline{U}$. Each such element is used a single time in the two substitutions, so that (ignoring array X and scalar YD) L = 1 and K = 2 in (3). If the LU array is on backing store, then this store is required to supply operands for (4) at M/2 words/sec.

This is a prohibitive rate, as evidenced by calculation of

γ = (arithmetic computation time)/(I/O time)

$\cong$ (chained multiply-subtract time)/(word transfer time)

$\cong 12.5\text{x}10^{-9}\text{m}/1.85\text{x}10^{-6}$

$\cong .0068\text{m}$

Thus, the processor will be busied less than 1 percent of the time with a single disc control, and less than 10% with full input channel activation.

This is a well-known problem in the solution of systems that cannot be contained in main memory; because of the Cray-1 processor

speed, it is simply aggravated. It is an interesting aspect of the problem that for the general class of sparse/full matrices, the above ratio is independent of matrix size, density, or equation ordering.

III. Evaluation of Processor-Memory Path

A. LU factorization

The data flow between the memory and processor is associated with the inner loops of the triangular factorization process. As a result, it becomes necessary to consider notation and details related to the LU factorization.

Let the matrix A be factored into

$$\underline{A} = \underline{L}\ \underline{U}$$

where

$$\underline{L} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ \ell_{21} & 1 & 0 & \cdots & 0 \\ \ell_{31} & \ell_{32} & 1 & \cdot & 0 \\ \vdots & \vdots & & \ddots & \vdots \\ \ell_{n1} & \ell_{n2} & & \cdots & 1 \end{bmatrix}, \quad \underline{U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & & \vdots \\ \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & & u_{nn} \end{bmatrix} \qquad (5)$$

One method of performing this decomposition on the Cray 1 was given in [2] and will be termed the inner product algorithm. It is described as follows.

At the k^{th} pivot step, assume that the u_{ij} has been computed for $1 \le i \le k-1$, $i \le j \le n$, and that ℓ_{ij} have been formed for $1 \le j \le k-1$, $j+1 \le i \le n$.

Then we calculate

$$\hat{\ell}_{ik} = a_{ik} - \overbrace{\sum_{m=1}^{k-1} \ell_{im} u_{mk}}^{\text{loop 1}} \qquad \overbrace{i = k+1,\ldots n}^{\text{loop 2}} \tag{6}$$

$$u_{kj} = a_{kj} - \overbrace{\sum_{m=1}^{k-1} \ell_{km} u_{mj}}^{\text{loop 3}} \qquad \overbrace{j = k,\ldots n}^{\text{loop 4}} \tag{7}$$

$$\ell_{ik} = \hat{\ell}_{ik}/u_{kk} \qquad i = k+1,\ldots n \tag{8}$$

where the loops are indicated in Figure 2. The inner product description follows from loops 1 and 3 above.

The inner product method requires accessing along both rows and columns and as such it may be necessary to skew the matrix in memory so that successive accesses are not made from the same memory bank.

The loop 1 portion of the inner product inner loop was implemented on the Cray-1 by loading into a vector register a portion of a row, $u_{1,m}$, $k \le m \le k+63$. This load was chained to a vector multiply involving $\ell_{k,1}$. The product was then chained to a vector subtract from the pre-fetched matrix elements $a_{k,m}$, $k \le m \le k+63$. This continued with

$$a_{k,m} = a_{k,m} - \ell_{k,2} * u_{2,m} \qquad k \le m \le k+63 \tag{9}$$

until

$$a_{k,m} = a_{k,m} - \ell_{k,k-1} * u_{k-1,m} \qquad k \le m \le k+63 \tag{10}$$

As can be seen, the u elements that participated in the vector multiply were streaming from main memory into the multiply pipe-line with final accumulation through the subtract pipeline. In this way up to 64 inner products are computed concurrently. The ℓ elements were computed similarly.

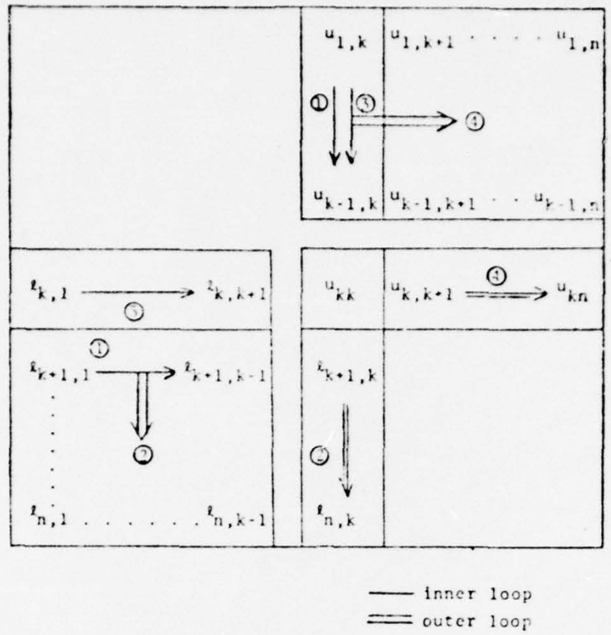


Figure 2. k^{th} pivot inner product calculation

In processing a matrix that is too large to be contained in main memory, there will be no time available for the I/O system to access memory while the inner loop is executing. As a result, the arithmetic and I/O operations cannot be concurrent and the factorization is slowed.

B. A reduced data flow algorithm

The alternative algorithm this report explores is based on a technique due to Pavkovich [3]. This algorithm employs the vector registers in such a way so as to reduce memory references to about 20% of what was required in the algorithm described above. This is accomplished by completing five rows of the matrix for each multiplicative use of a U vector rather than completing only a single row. This should leave the I/O system sufficient memory access time to allow the movement of portions of the matrix to and from the disk without slowing the functional units.

The arithmetic computation for this algorithm is numerically identical to what was described above. The sequence of operations is changed with the effect being a block-wise reduction of the matrix. With reference to Figure 3, the following sequence is performed at the k^{th} block step.

1) Reduce the q x q diagonal block comprising elements $a_{i,j}$, $k \leq i \leq k+q-1$, $k \leq j \leq k+q-1$.

2) Reduce the q x p row blocks. The last row block may have fewer than p column elements.

3) Reduce the p x q column blocks. The last column block may have fewer than p row elements.

4) Repeat steps 1-3 with the remaining k-q x k-q sub-matrix.

5) The factorization will be complete with the reduction of the south-east corner diagonal block. This block may have fewer than q row and column elements.

For the present, only the reduction of the q x p row blocks and the p x q column blocks will be discussed.

The structure of the off-diagonal blocks and the matrix traversal scheme was chosen with the algorithm's implementation on the Cray-1 in mind. In this way one is always dealing with full length vectors except at a row or column end. The detail of a row block appears in Figure 4.

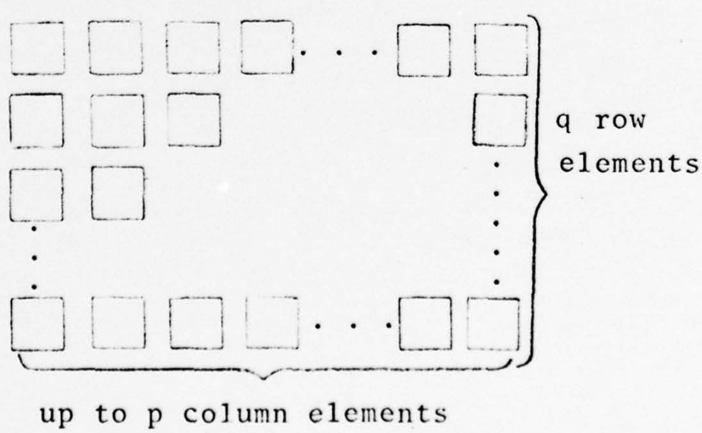


Figure 4. Row block detail

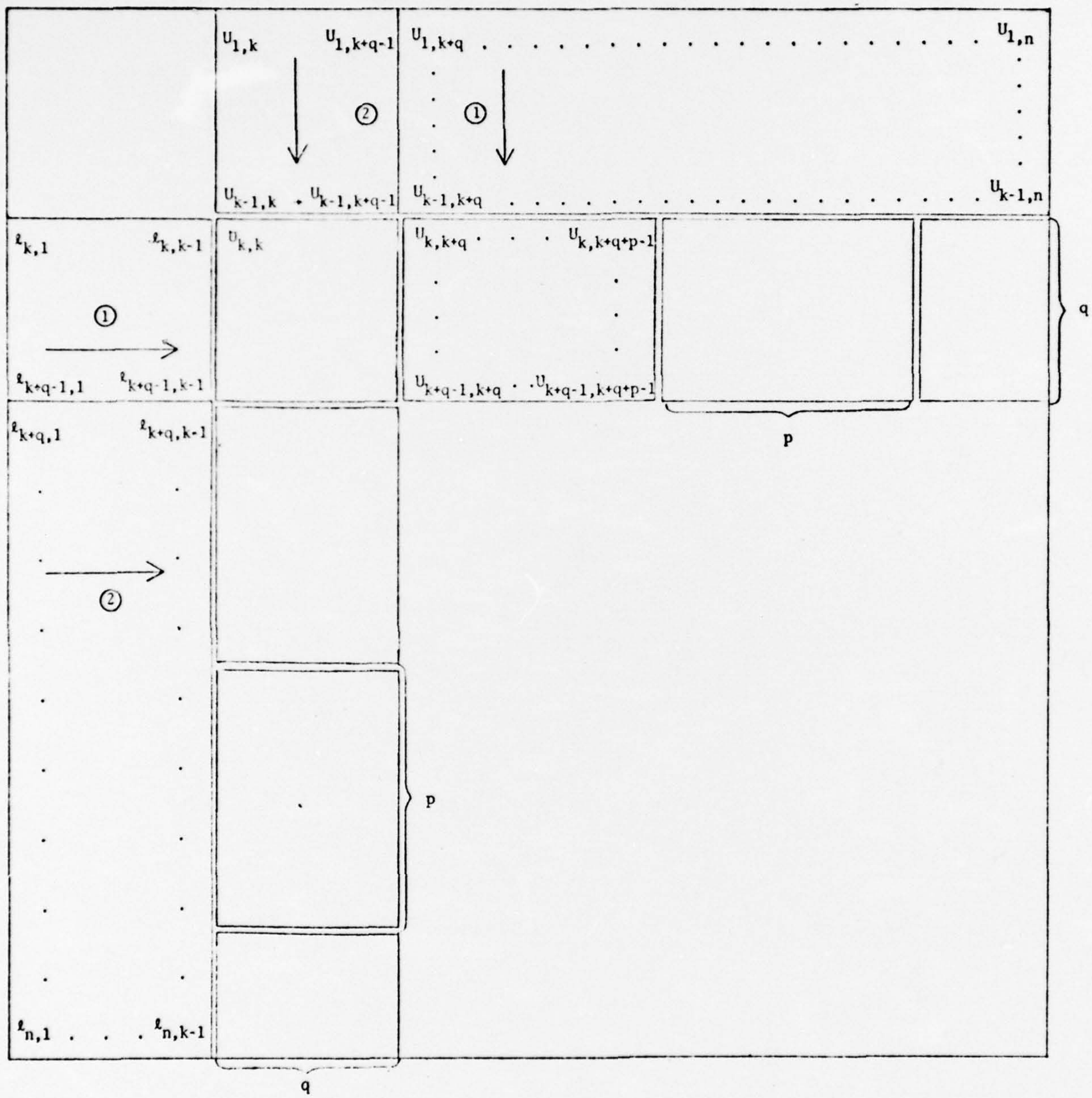


Figure 3. K^{th} pivot step block reduction.

C. Row block reduction

This block is read from memory into the vector registers of the Cray-1. As a result, the limitation on p is 64 column elements, this being the number of storage locations in a vector register. With eight vector registers available, three of which are used to perform the inner product calculation, q has the value of five.

The processing of a typical row block proceeds as follows. First the q x p block to be reduced is loaded into the q vector registers set aside for that purpose. These vector registers will be denoted $\overline{B}_1$ to $\overline{B}_q$. The inner loop operation sequence is carried out for $1 \leq i \leq k-1$ (see Figure 5).

1) Load RV_i into the row vector register, $\overline{V}_{rv}$.
2) Load SG_i into the scalar group registers, S_j, $1 \leq j \leq q$.
3) With the data now present in registers, q computation sequences can be carried out. Each computation sequence is of the form

$$\overline{B}_m = \overline{B}_m - S_m \times \overline{V}_{rv}, \quad 1 \leq m \leq q.$$

The computation represented by this equation is performed element by element on each of the p components of the vector register.

This completes all computation external to the reduction block. Consequently, only the $\overline{B}_1$ block vector is completely reduced, with $\overline{B}_2 \ldots \overline{B}_q$ not yet finished. The reduction is completed by using reduced $\overline{B}$ vectors internal to the block to reduce the balance of the block.

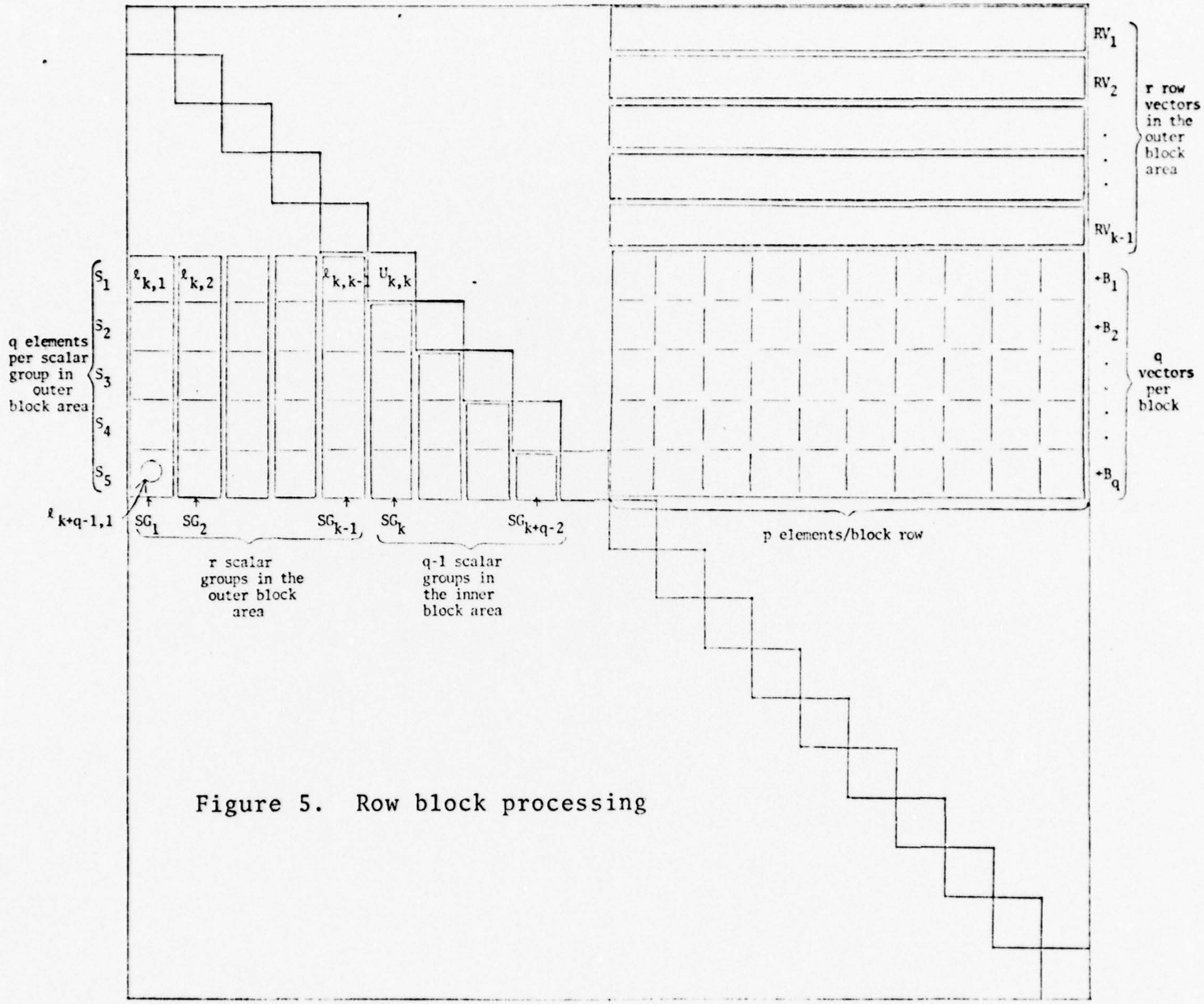


Figure 5. Row block processing

The following sequence is carried out for $1 \leq i \leq q-1$ to complete the block reduction.

1) Load SG_{k+i-1} into the scalar group registers, S_j, $i+1 \leq j \leq q$.
2) With the data now present in registers, $q-i$ computation sequences can be done. Each computation sequence is of the form

$$\overline{B}_m = \overline{B}_m - S_m \times \overline{B}_i \ , \ i+1 \leq m \leq q.$$

At the i^{th} completion of this sequence, $\overline{B}_{i+1}$ is fully reduced and can be returned to memory and also used to complete the remaining $\overline{B}$ vectors.

D. Accounting for memory access reduction

To justify the claim that main memory accesses are reduced using this algorithm it is necessary to determine the ratio (ρ) of multiplies (additions are essentially free) to main memory accesses. Table 1 delineates the quantities involved. The quantity r is the number of row vectors external to the reduction block. The ratio is then

$$\rho = \frac{q/2(q+2r-1)p \qquad \text{arithmetic operations}}{q/2(q+2r-1)+(r+2q)p \qquad \text{memory accesses}}$$

Scalar fetches: $qr + \frac{q(q-1)}{2}$
Vector element fetches: $(r+q)p$
Vector element stores: qp
Multiplies required: $[qr + \frac{q(q-1)}{2}]p$

Table 1. Ratio quantities

The number of multiplies necessary is essentially a function of the number of scalars involved, which is primarily a function of r since q is fixed in the implementation. Asymptotically, this ratio becomes

$$\rho_a = \lim_{r \to \infty} \rho = \frac{qp}{q+p} \text{ operations/access}$$

Also note that as p, which is the vector register length, increases

$$\lim_{p \to \infty} \rho_a = q \text{ operations/access}$$

we asymptotically can achieve no better than q arithmetic operations per memory access, indicating that longer vector registers would not improve this aspect of the algorithm.

Using values for q = 5 and p = 64 as in the Cray-1 implementation, the asymptotic performance is $\rho = 4.63$. As the vector register length increases the effect of the q scalar loads diminishes and ρ asymptotes to 5 operations/(memory access).

Contrasted with q=1 and p=64 used in the original inner product algorithm where ρ=.984 operations/(memory access), the improved algorithm yields a 78.7% reduction in memory accesses.

E. Algorithm implementation on the Cray-1

The data flow of the inner loop, which performs the block reduction, is quite straight forward. Figure 6a illustrates the data flow through the vector registers as the processing, external to the block, is carried out.

The physical vector registers in the Cray-1 are assigned names V0 through V7. The contents of each vector register has been assigned a label with an overbar indicating what is currently stored there, (i.e., $\overline{B}_1$ through $\overline{B}_5$ refer to the five vectors comprising the reduction block, $\overline{RV}_i$ refers to the ith row vector residing in a vector register,etc).

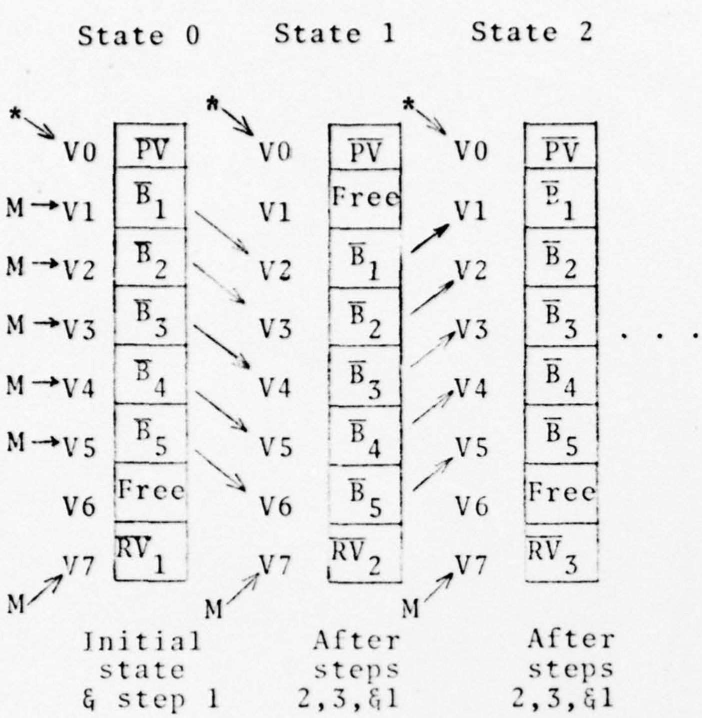


Figure 6a. Data flow in external block processing

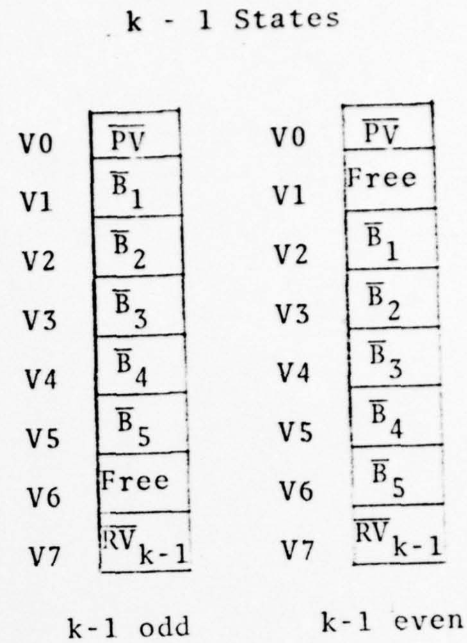


Figure 6b. Two possible end states after external block processing is complete

In the allocation of the vector registers, six (V1-V6) are assigned to the reduction block, of which five contain the actual data of the block (q=5) with a sixth (labeled free) being used to allow accumulation into the block of $\overline{B}$ vectors. One of the remaining two vector registers (V7) is used for holding the row vectors RV_i, $1 \leq i \leq k-1$ as they are read one at a time from memory. The other (V0) is used to contain the product vector $\overline{PV}$ which is computed from $\overline{RV}_i$ by a scalar multiplication.

When all external k-1 row vectors are processed, the contents of the vector registers will be in one of two states as depicted in Figure 6b. Which k-1 end state results is governed by whether the number of k-1 row vectors processed is odd or even.

There are two similar code sequences used in processing row vectors:

1) accumulation of the $\overline{B}_1$-$\overline{B}_5$ block into vector registers V2-V6;
2) accumulation of the block into vector registers V1-V5. This accumulation is depicted in Figure 6a by " $\overset{+}{\rightarrow}$ " symbol. Initially vector registers V1-V5 must be loaded with the reduction block from main memory, which is depicted by the " M→ " symbol. Then the first row vector RV_1 (Figure 5) must be loaded into V7 from main memory. Then the last scalar of scalar group one (SG_1) is loaded and the product of $\overline{RV}_1$ and the scalar is placed in V0 ($\overline{PV}$) which is depicted by the " *→ " symbol. This product is then accumulated with $\overline{B}_5$ (in V5) and directed to V6. This row vector is then used with the remaining four scalars working up the scalar group. This leaves V1 free to propagate results in the reverse direction with the next row vector.

What follows is the symbolic layout of vector instructions that reflect the operation sequence of the external row block processing (refer to Figures 5 and 6a). Only the pertinent instructions have been included. Scalar code necessary for address generation and loop control have been omitted and the braces indicate chained instructions.

```
                        i←1                                Initialize row vector and scalar
                                                           group counter
                        Load V1,V2,V3,V4,V5 with B1,B2,B3,B4,B5, respectively
                           ↓
ROW_BLK_LOOP            Load SG_i                          load the i^th scalar group

                        ⎧V7←RV_i                           load the i^th row vector
                        ⎨V0←S_5*V7                         create B̄_5 product vector
                        ⎩V6←V5-V0                          accumulate B̄_5

                        ⎧V0←S_4*V7                         create B̄_4 product vector
                        ⎩V5←V4-V0                          accumulate B̄_4

                        ⎧V0←S_3*V7                         create B̄_3 product vector
                        ⎩V4←V3-V0                          accumulate B̄_3

                        ⎧V0←S_2*V7                         create B̄_2 product vector
                        ⎩V3←V2-V0                          accumulate B̄_2

                        ⎧V0←S_1*V7                         create B̄_1 product vector
                        ⎩V2←V1-V0                          accumulate B̄_1

                           ↓

          If i = k-1 JUMP TO ODD_ROW_DRAIN                 check for completion of external row
                                                           vectors
                        i←i+1                              move on to the next one
                           ↓
                        Load SG_i                          load the i^th scalar group

                        ⎧V7←RV_i                           load the i^th row vector
                        ⎨V0←S_1*V7                         create B̄_1 product vector
                        ⎩V1←V2-V0                          accumulate B̄_1

                        ⎧V0←S_2*V7                         create B̄_2 product vector
                        ⎩V2←V3-V0                          accumulate B̄_2

                        ⎧V0←S_3*V7                         create B̄_3 product vector
                        ⎩V3←V4-V0                          accumulate B̄_3

                        ⎧V0←S_4*V7                         create B̄_4 product vector
                        ⎩V4←V5-V0                          accumulate B̄_4

                        ⎧V0←S_5*V7                         create B̄_5 product vector
                        ⎩V5←V6-V0                          accumulate B̄_5

                           ↓

          If i = k-1 JUMP TO EVEN_ROW_DRAIN                check for completion of external row
                                                           vectors
                        i←i+1                              move on to the next one

                        JUMP TO ROW_BLK_LOOP
```

The column block reduction uses a similar scheme with the only alteration being where in memory the matrix data is coming from.

At this stage of the algorithm all the external row vectors have been processed with the contents of the vector registers in one of the two k-1 states depicted in Figure 6b. As discussed earlier, the rest of the row vectors come from inside the reduction block to complete the blocks reduction. In fact, $\overline{B}_1$ is now completely reduced and will be employed as the next row vector to further reduce $\overline{B}_2$, $\overline{B}_3$, $\overline{B}_4$ and $\overline{B}_5$. Figures 7a and b show the internal block reduction data flow starting from either of the k-1 ending states of the external block reduction. This internal block reduction is referred to as draining the block.

Figure 7a. Internal block reduction data flow after an odd number of external row vectors processed. (k-1 odd)

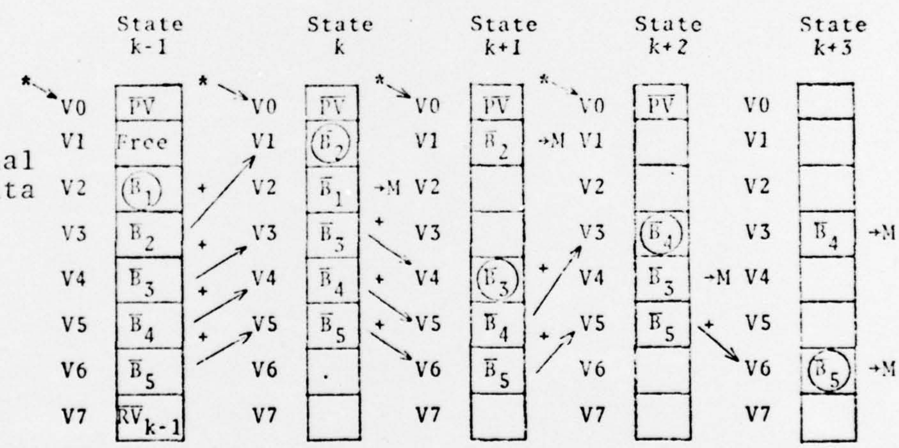
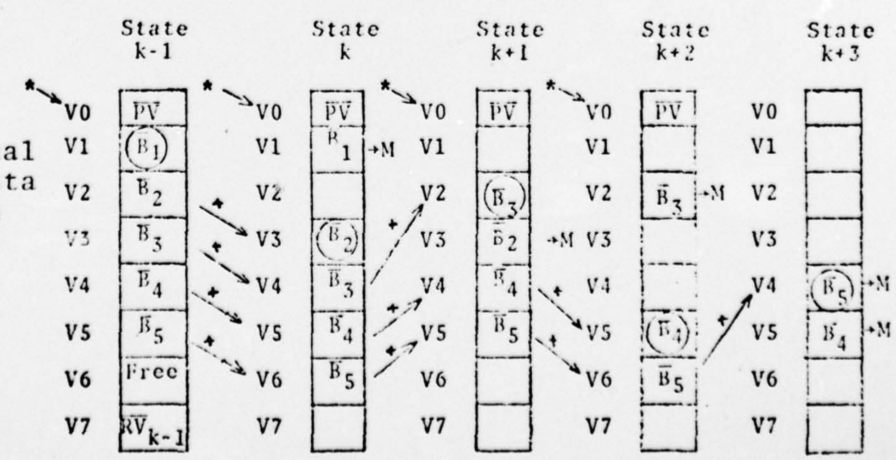


Figure 7b. Internal block reduction data flow after an even number of external row vectors processed. (k-1 even)



At each drain state the completion of a $\overline{B}$ vector is indicated by a circle. In the subsequent states, as the rest of the $\overline{B}$ vectors are completed they are successively returned to memory ("→M symbolizes the store of the vector register to memory). The odd and even k-1 states give rise to two block drain codes.

What follows is the symbolic layout of vector instructions necessary for the draining of a row block starting from the odd k-1 state. This data flow is depicted in Figure 7a. Also refer to Figure 5. The draining starting from the even k-1 state is similar in spirt as is the draining of a column reduction block. Again, scalar code for address generation has been omitted for the sake of clarity and the braces indicate chained or concurrent instruction sequences.

```
ODD_DRAIN   i←i+1            Move to next scalar group (i=k-1 at entry)
            Load SG_i        Load the scalar group
            V0←S_2*V2        Create B̄_2 product vector
            V1←V3-V0         Accumulate B̄_2 (B̄_2 now fully reduced)
            V0←S_3*V2        Create B̄_3 product vector
            V3←V4-V0         Accumulate B̄_3
            V0←S_4*V2        Create B̄_4 product vector
            V4←V5-V0         Accumulate B̄_4
            V0←S_5*V2        Create B̄_5 product vector
            V5←V6-V0         Accumulate B̄_5
            Store V2 containing B̄_1 to memory (optional pivot before store)
            i←i+1            Move to next scalar group
            Load SG_i
            V0←S_5*V1        Create B̄_5 product vector
            V6←V5-V0         Accumulate B̄_5
            V0←S_4*V1        Create B̄_4 product vector
            V5←V4-V0         Accumulate B̄_4
            V0←S_3*V1        Create B̄_3 product vector
            V4←V3-V0         Accumulate B̄_3 (B̄_3 now fully reduced)
            Store V1 containing B̄_2 to memory (optional pivot before store)
            i←i+1            Move to next scalar group
            Load SG_i
            V0←S_4*V4        Create B̄_4 product vector
            V3←V5-V0         Accumulate B̄_4 (B̄_4 now fully reduced)
            V0←S_5*V4        Create B̄_5 product vector
            V5←V6-V0         Accumulate B̄_5
            Store V4 containing B̄_3 to memory (optional pivot before store)
            i←i+1            Move to next scalar group
            Load SG_i
            V0←S_5*V3        Create B̄_5 product vector
            V6←V5-V0         Accumulate B̄_5 (B̄_5 now fully reduced)
            Store V3 containing B̄_4 to memory (optional pivot before store)
            Store V6 containing B̄_5 to memory (optional pivot before store)
            Row block reduction now complete and returned to memory.
```

Once all the row vectors comprising the block have been returned to memory the block reduction is complete. Depending on whether row or column pivoting is desired an optional pivot multiplication can be inserted before the vector stores in the appropriate drain routine.

Since there is only one path to the Cray-1 main memory from the computational section the scalar fetches should be placed in such a way so as to avoid conflicting with the block vector stores and thus causing a halt to instruction issuing.

It is easy to see that when a row vector is read from main memory each element is employed in five multiplies giving rise to the asymptotic limit of $\rho = 5$ as $p \to \infty$.

IV. Commentary

A. Further algorithmic considerations

The problem of pivoting has been ignored in this report, although it involves data restructuring and hence can influence data flow. It would be considered when the more general problem of data formatting was being investigated. Here, one would be concerned with the formatting of $\underline{A}$, $\underline{L}$, and $\underline{U}$ in the disc, main memory, and vector registers.

Since the disc is read and written sequentially it must share at least a common block format with main memory. However, the data may be reformatted between memory and the registers, since a matrix block in memory may be accessed both row- and column-wise. Also, $\underline{A}$ and $\underline{L}$ $\underline{U}$ need not be identically formatted; the first may be determined by user convenience whereas $\underline{L}$ and $\underline{U}$ are internal to the equation solution algorithm. The result may be that the alternate row- and column-strip access of Figure 3 may be inconvenient; this would not seriously impact the interior loop where the reduced memory accesses are achieved.

B. Speedup by architectural modifications

1. Introduction

With the processor-memory path busied between 1/4 and 1/5 of the time in support of the inner loops, the memory is free to communicate with the instruction parcel buffers, the B and T registers, and (most importantly) the backing store the majority of the time.

We have shown that lack of disc capacity can impede computation

overall; however even if solution time becomes I/O dominated so that I/O traffic is maximized, it can be readily shown that memory will be occupied at most m/144 of the time emptying I/O buffers. Adding this to the above fractions, it seems safe to assume that 1/2 - 2/3 of the memory bandwidth is unused for all purposes presently accounted for.

In the following sections, we suggest architectural modifications to utilize this bandwidth to speedup the equation solution.

2. Register control

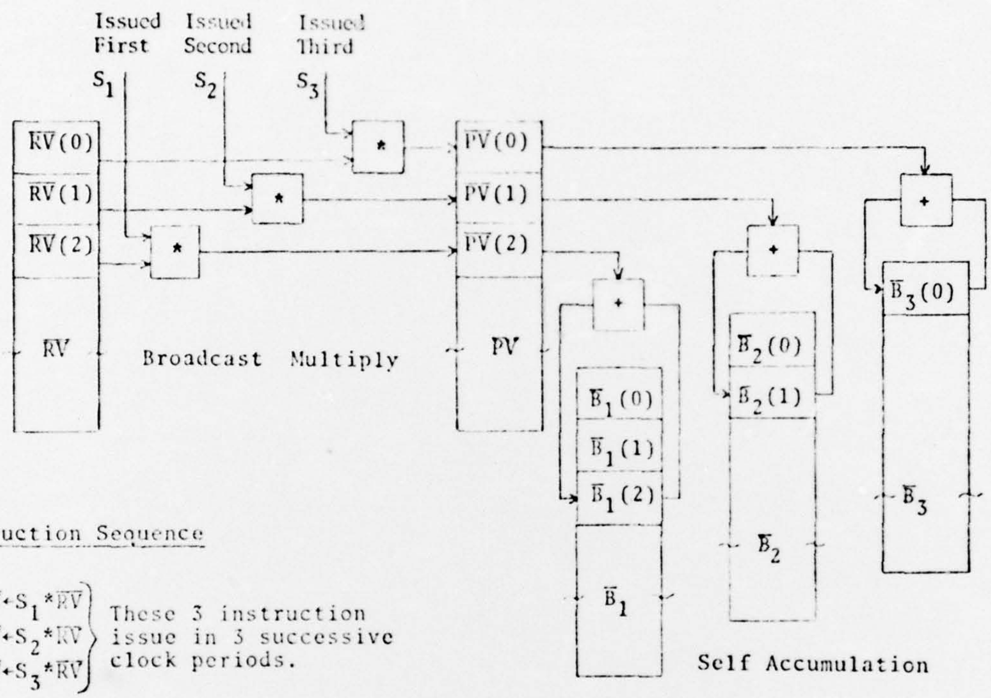
Presently, access to elements of a vector register are controlled by the register with a single counter. If the access control were moved from the vector register to the functional unit, the same vector register could be concurrently accessed by multiple functional units. This would permit self accumulation of a vector register into itself, allowing, for example, incrementation as $V1 \leftarrow V1 + C$. The register propagation scheme of the proposed algorithm where one vector register was added to another could be avoided, allowing a q of 6 and a further reduction in memory accesses.

3. Expanded functional units

A more important consequence of excess memory bandwidth and modified register control would be the ability to add functional units for increased parallelism. If each multiply-add unit required the same fraction of memory bandwidth as determined above, possibly two or three units could be added without interference. One possibility for accomplishing this without changing

instruction formats and thereby making present codes upward compatible is now described.

The data flow is symbolically diagrammed in Figure 8 with the vector instruction sequence that produced it.



Instruction Sequence

$\overline{PV} \leftarrow S_1 * \overline{RV}$
$\overline{PV} \leftarrow S_2 * \overline{RV}$
$\overline{PV} \leftarrow S_3 * \overline{RV}$
These 3 instruction issue in 3 successive clock periods.

$\overline{B}_1 \leftarrow \overline{B}_1 - \overline{PV}$ This issues at the chain slot arrival of data from the S_1 multiply.
$\overline{B}_2 \leftarrow \overline{B}_2 - \overline{PV}$ This issues at the next chain slot from the S_2 multiply.
$\overline{B}_3 \leftarrow \overline{B}_3 - \overline{PV}$ Issues at S_3 multiply chain slot.

Figure 8. Symbolic data flow of parallel inner product.

Essentially, three inner products are computed in parallel with a clock period displacement in the elements being processed. The first multiply issued would be one clock period ahead of the second one issued. The second in turn would be one clock period ahead of the third. The first product to arrive at $\overline{PV}(0)$ (The notation $\overline{PV}(0)$ refers to element zero for vector register $\overline{PV}$) would signal chain slot time for the waiting $\overline{B}_1$ accumulation instruction. This

instruction would issue with the product being fed into the adder. In the next clock period $\overline{PV}(1)$ would receive the first multiply's second product which would move on to $\overline{B}_1$'s adder, and so forth. Also, in this clock period, $\overline{PV}(0)$ would receive the second multiply's first product thereby signaling chain slot time for the now waiting $\overline{B}_2$ accumulation instruction, causing it to issue. In the next clock period the third multiply's first product will arrive at $\overline{PV}(0)$ initiating the $\overline{B}_3$ accumulation. In succeeding clock periods, three products will simultaneously arrive at $\overline{PV}$ and then be passed on to the three adders for $\overline{B}$ vector accumulation.

This scheme is not without its drawbacks. The ordering and placement of vector instructions has to be critically observed noting all instruction issue delays to insure correct chain slot time synchronization. When using a vector register ($\overline{PV}$) in this multi-chain through mode, missing a chain slot time would produce an entirely different calculation. Furthermore, external interupts (I/O, timer, etc.) would have to be disallowed during a multi-store use of any vector register, thereby anticipating the invocation of multi-chain through mode by subsequent, as yet unseen, vector instructions.

4. Scalar register chaining

When intermediate results of a chained computation are not required, it is undesireable to allow them to occupy a vector register. One technique might be to allow chaining through scalar registers. When a vector result is passed to a chain through

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR - TR - 76 - 1212	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) DATA FLOW CONSIDERATIONS IN IMPLEMENTING A FULL MATRIX SOLVER WITH BACKING STORE ON THE CRAY-1		5. TYPE OF REPORT & PERIOD COVERED Interim rept.,
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) D. A. Orbits, D. A. Calahan		8. CONTRACT OR GRANT NUMBER(s) AF-AFOSR-2812-75
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Michigan, Dept of Electrical & Computer Engineering, Ann Arbor, Michigan 48109		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F A3 2304/A3
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research (NM) Bolling AFB, DC 20332		12. REPORT DATE 1976
		13. NUMBER OF PAGES 28
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) 1 Sep 76 32p.		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

98

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Vector processors
Parallel processors
Matrix analysis

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Techniques for the solution of full systems of simultaneous equations represent an important algorithm class for vector processors. This report considers the data flows involved in solving a full equation system on the Cray-1. This involves study of the I/O and memory-processor path traffic vis-a-vis the capabilities of the Cray-1 to support it. The I/O is found to present problems for small and large systems. Using an algorithm proposed in the report, the memory-processor path is shown to have excess bandwidth. Suggestions are made for utilizing this bandwidth to increase the arithmetic operation rate by

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE UNCLASSIFIED 407 400

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Block 20/Abstract

modifying and expanding the processor architecture.

scalar register, it would simply move on to the next functional unit of the chained instruction sequence. The following hypothetical instruction sequence is an example of using a scalar register in this manner.

$S1 \leftarrow S2*\overline{RV}$	Begin production of product elements
$V6 \leftarrow V5+S1$	Accumulate these products

References

[1] Cray-1 Computer System Reference Manual, No. 2240004, Cray Research, Inc.

[2] Calahan, D. A., W. N. Joy, D. A. Orbits, "Preliminary Report on Results of Matrix Benchmarks on Vector Processors," Report #94, Systems Engineering Laboratory, University of Michigan, May, 1976.

[3] Pavkovich, J. M., "The Solution of Large Systems of Algebraic Equations," Stanford University Computer Science Department, RPT. CS-2, December 6, 1963. (Available from NTIS as AD 427 753).